The Dirty-Block Index

Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu,
Phillip B. Gibbons†, Michael A. Kozuch†, Todd C. Mowry

Carnegie Mellon University †Intel Pittsburgh

Abstract

On-chip caches maintain multiple pieces of metadata about each cached block—e.g., dirty bit, coherence information, ECC. Traditionally, such metadata for each block is stored in the corresponding tag entry in the tag store. While this approach is simple to implement and scalable, it necessitates a full tag store lookup for any metadata query—resulting in high latency and energy consumption. We find that this approach is inefficient and inhibits several cache optimizations.

In this work, we propose a new way of organizing the dirty bit information that enables simpler and more efficient implementations of several optimizations. In our proposed approach, we remove the dirty bits from the tag store and organize it differently in a separate structure, which we call the Dirty-Block Index (DBI). The organization of DBI is simple: it consists of multiple entries, each corresponding to some row in DRAM. A bit vector in each entry tracks whether or not each block in the corresponding DRAM row is dirty.

We demonstrate the benefits of DBI by using it to simultaneously and efficiently implement three optimizations proposed by prior work: 1) Aggressive DRAM-aware writeback, 2) Bypassing cache lookups, and 3) Heterogeneous ECC for clean/dirty blocks. DBI, with all three optimizations enabled, improves performance by 31% compared to the baseline (by 6% compared to the best previous mechanism) while reducing overall cache area cost by 8% compared to prior approaches.

1. Introduction

On-chip caches in modern processors maintain multiple pieces of metadata about cached blocks—e.g., dirty bit, coherence state, ECC. Traditionally, caches store such metadata for each block in the corresponding tag entry in the tag store. Although this approach is straightforward and scalable, even simple metadata queries require a full tag store lookup, which incurs high latency and energy [33]. In this work, we focus our attention on the dirty bit information in writeback caches.

We find that the existing approach of organizing the dirty bit information in the tag entry inhibits several cache optimizations. More specifically, we find that several cache optimizations require the cache to quickly and efficiently 1) determine if a block is dirty [33, 44, 49], and 2) identify the list of all spatially co-located dirty blocks—i.e, dirty blocks from the same DRAM row [27, 47, 51]. However, checking the dirty status of even a single cache block with the existing cache organization requires an expensive tag store lookup.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

In this work, we propose a new way of organizing the dirty bit information. In our proposed organization, we remove the dirty bits from the cache tag entries and organize them differently in a separate structure, which we call the Dirty-Block Index (DBI).[1] All queries regarding the dirty status are directed to the DBI. The organization of the DBI is simple. It has multiple entries, each tracking the dirty bit information of all the blocks in some DRAM row. Each entry contains a row tag indicating which DRAM row the entry corresponds to, and a bit vector indicating whether each block in the DRAM row is dirty. A cache block is dirty *if and only if* the DBI has a valid entry corresponding to the DRAM row containing the block and the bit corresponding to the cache block in the associated bit vector is set.

DBI has three nice properties. First, since it is much smaller than the main tag store, it can identify whether a block is dirty much faster than the main tag store. Second, since the dirty bit information of all blocks of a DRAM row are stored together, DBI can list all dirty blocks of a DRAM row with a single query (whereas the main tag store requires one query for each block of the row). Finally, since DBI is organized independently of the main tag store, it can be used to limit the number of dirty blocks to a small fraction of the number of blocks in the cache. These properties allow DBI to efficiently implement and accelerate several cache optimizations. In this work, we quantitatively evaluate the benefits of DBI using three previously proposed optimizations.

First, prior works [27, 51] have shown that proactively writing back all dirty blocks of a DRAM row in a single burst performs significantly better compared to writing them back in the order that they are evicted from the cache. DBI allows the cache to efficiently identify all dirty blocks of a DRAM row, enabling a simpler implementation of this optimization. Second, prior works [33, 44] have proposed mechanisms to bypass the cache lookup for accesses that are likely to miss in the cache. However, since accesses to dirty blocks cannot be bypassed, previously proposed implementations of this optimization incur high complexity. In contrast, DBI, with its ability to quickly identify if a block is dirty, enables a very simple and efficient implementation of this optimization. Third, prior works [30, 58, 59] have proposed mechanisms to reduce the overhead of ECC in caches by storing only a simple error detection code for each clean block and a strong error correction

[1] Many cache coherence protocols may maintain the dirty status implicitly in the cache coherence states. Section 2.3 discusses how different cache coherence protocols can be seamlessly adapted to work with DBI.

code for each dirty block. However, since any block in the cache can be dirty, prior approaches require complex changes to implement this optimization. In our proposed organization, since DBI decouples the dirty bit information from the tag store, it is sufficient to maintain strong ECC for only blocks tracked by the DBI. Section 3 discusses these optimizations and our proposed implementations using DBI.

We compare DBI (with different combinations of these optimizations) to two baseline mechanisms (a cache employing the Least Recently Used policy and one employing the Dynamic Insertion Policy [18, 42]), and three previous mechanisms that employ individual optimizations (DRAM-aware writeback (DAWB) [27], Virtual Write Queue (VWQ) [51], and Skip Cache [44]). The results show that DBI, with all optimizations enabled, outperforms all prior approaches (6% compared to the best previous mechanism and 31% compared to the baseline for a 8-core system) while also significantly reducing overall cache area (8% compared to the baseline for a 16MB cache). Section 6 discusses these results in more detail.

While we discuss three optimizations enabled by DBI in detail, DBI can be used to enable several other mechanisms. Section 7 briefly describes some of these mechanisms.

The main contributions of this paper are as follows.

- We propose a new way of tracking dirty blocks that maintains dirty bit information of cache blocks in a separate structure called the Dirty-Block Index (DBI), instead of in the main tag store.
- We show that our new dirty-bit organization using DBI enables efficient implementation of several previously proposed optimizations that involve dirty blocks. DBI simultaneously enables all these optimizations using a single structure whereas prior approaches require separate structures for each optimization.
- We quantitatively compare the performance and area benefits of DBI using three optimizations to prior approaches [18, 27, 44, 51]. Our evaluations show that DBI with all optimizations outperforms all previous mechanisms while reducing cache area cost. The performance improvement is consistent across a wide variety of system configurations and workloads.

2. The Dirty-Block Index (DBI)

We conceived the idea of the Dirty-Block Index based on two key principles motivated by two previously proposed optimizations related to dirty blocks: cache lookup bypass [33, 44] and DRAM-aware writeback [27, 51]. First, prior works have shown that bypassing the cache lookup for an access that is likely to miss in the cache can reduce average latency and energy consumed by memory accesses [33]. However, the cache *must not* be bypassed for a dirty block. This motivates the need for a mechanism that can quickly check if a block is dirty (without looking up the entire tag store). Second, prior works have shown that writing back spatially co-located dirty blocks (i.e., those that belong to the same DRAM row [45, 60]) together improves system performance significantly. This motivates the need for a mechanism to quickly and efficiently identify spatially co-located dirty blocks.

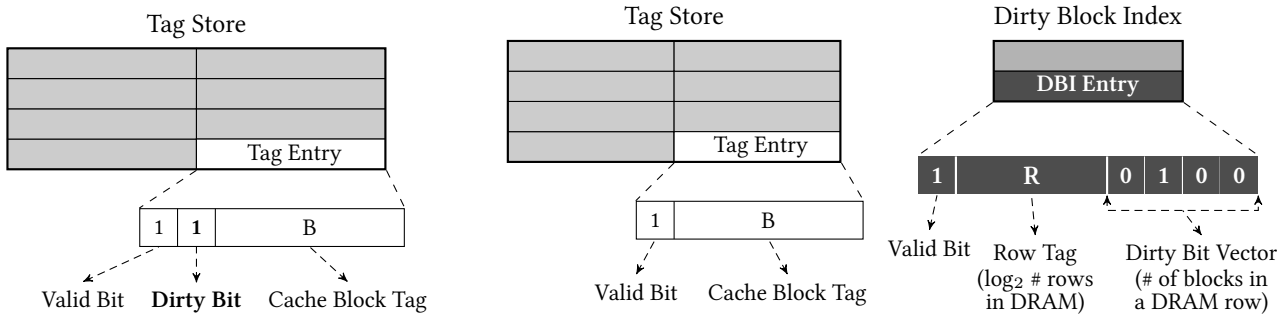
Based on these principles, we propose to remove the dirty bits from the main tag store and store them in a separate structure, called the Dirty-Block Index (DBI). DBI reorganizes the dirty bit information such that the dirty bits of blocks of the same DRAM row are stored together in a single entry.

2.1. DBI Structure

Figure 1 compares the conventional tag store organization with a tag store augmented with a DBI. In the conventional organization (shown in Figure 1a), each tag entry contains a dirty bit that indicates whether the corresponding block is dirty or not. For example, to indicate that a block B is dirty, the dirty bit of the corresponding tag entry is set.

In contrast, in a cache augmented with a DBI (Figure 1b), the dirty bits are removed from the main tag store and organized differently in the DBI. The organization of DBI is simple. It consists of multiple entries. Each entry corresponds to some row in DRAM—identified using a *row tag* present in each entry. Each DBI entry contains a *dirty bit vector* that indicates if each block in the corresponding DRAM row is dirty or not.

DBI Semantics. A block in the cache is dirty *if and only if* the DBI contains a valid entry for the DRAM row that contains the block and the bit corresponding to the block in the



(a) Conventional cache tag store

(b) Cache tag store augmented with a DBI

Figure 1: Comparison between conventional cache organization and a cache augmented with a DBI. The figure shows how each organization represents the dirty status of a block B. Block B is assumed to be the second block in the DRAM row R.

bit vector of that DBI entry is set. For example, assuming that block B is the second block of DRAM row R, to indicate that block B is dirty, the DBI contains a valid entry for DRAM row R, with the second bit of the corresponding bit vector set.[2]

2.2. DBI Operation

Figure 2 pictorially describes the operation of a cache augmented with a DBI. The focus of this work is on the on-chip last-level cache (LLC). Therefore, for ease of explanation, we assume that the cache does not receive any sub-block writes and any dirty block in the cache is a result of a writeback generated by the previous level of cache.[3] There are four possible operations, which we describe in detail below.

2.2.1. Read Access to the Cache

The addition of DBI *does not change* the path of a read access in any way. On a read access, the cache simply looks up the block in the tag store and returns the data on a cache hit. Otherwise, it forwards the access to the memory controller.

2.2.2. Writeback Request to the Cache

In a system with multiple levels of on-chip cache, the LLC will receive a writeback request when a dirty block is evicted from the previous level of cache. Upon receiving such a writeback request, the cache performs two actions (as shown in Figure 2). First, it inserts the block into the cache if it is not already present. This may result in a cache block eviction (discussed in Section 2.2.3). If the block is already present in the cache, the cache just updates the data store (not shown in the figure) with the new data. Second, the cache updates the DBI to indicate that the written-back block is dirty. If the DBI already has an entry for the DRAM row that contains the block, the cache simply sets the bit corresponding to the block in that DBI entry. Otherwise, the cache inserts a new entry into the DBI for the DRAM row containing the block and with the bit corresponding to the block set. Inserting a new entry into the DBI may require an existing DBI entry to be evicted. Section 2.2.4 discusses how the cache handles such a DBI eviction.

2.2.3. Cache Eviction

When a block is evicted from the cache, it has to be written back to main memory if it is dirty. Upon a cache block eviction, the cache consults the DBI to determine if the block is dirty. If so, it first generates a writeback request for the block and sends it to the memory controller. It then updates the DBI to indicate that the block is no longer dirty—done by simply resetting the bit corresponding to the block in the bit vector of the DBI entry. If the evicted block is the last dirty block in the corresponding DBI entry, the cache invalidates the DBI entry so that the entry can be used to store the dirty block information of some other DRAM row.

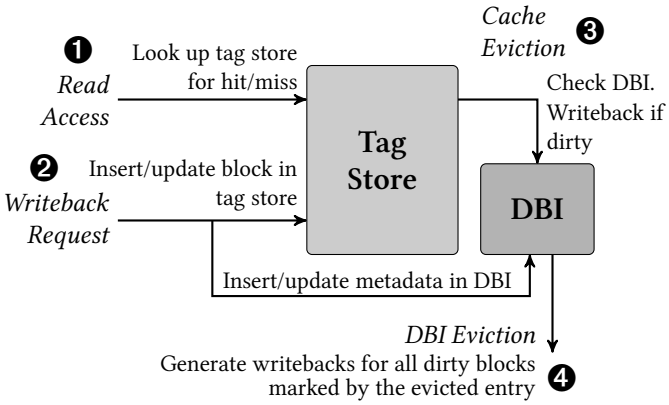


Figure 2: Operation of a cache with DBI

2.2.4. DBI Eviction

The last operation in a cache augmented with a DBI is a *DBI eviction*. Similar to the cache, since the DBI has limited space, it can only track the dirty block information for a limited number of DRAM rows. As a result, inserting a new DBI entry (on a writeback request, discussed in Section 2.2.2) may require evicting an existing DBI entry. We call this event a *DBI eviction*. The DBI entry to be evicted is decided by the DBI replacement policy (discussed in Section 4.3). When an entry is evicted from the DBI, *all* the blocks indicated as dirty by the entry should be written back to main memory. This is because, once the entry is evicted, the DBI can no longer maintain the dirty status of those blocks. Therefore, not writing them back to memory will likely lead to incorrect execution, as the version of those blocks in memory is stale. Although a DBI eviction may require evicting many dirty blocks, with a small buffer to keep track of the evicted DBI entry (until all of its blocks are written back to memory), the DBI eviction can be interleaved with other demand requests. Note that on a DBI eviction, the corresponding cache blocks need not be evicted from the cache—they only need to be transitioned from the dirty state to clean state.

2.3. Cache Coherence Protocols

Many cache coherence protocols implicitly store the dirty status of cache blocks in the cache coherence states. For example, in the MESI protocol [37], the M (modified) state indicates that the block is dirty. In the improved MOESI protocol [52], both M (modified) and O (Owner) states indicate that the block is dirty. To adapt such protocols to work with DBI, we propose to split the cache coherence states into multiple pairs—each pair containing a state that indicates the block is dirty and the non-dirty version of the same state. For example, we split the MOESI protocol into three parts: (M, E), (O, S) and (I). We can use a single bit to then distinguish between the two states in each pair. This bit will be stored in the DBI.

[2]Note that the key difference between the DBI and the conventional tag store is the *logical organization* of the dirty bit information. While some processors store the dirty bit information in a separate physical structure, the logical organization of the dirty bit information is same as the main tag store.

[3]Sub-block writes typically occur in the primary L1 cache where writes are at a word-granularity, or at a cache which uses a larger block size than the previous level of cache. The DBI operation described in this paper can be easily extended to caches with sub-block writes.

3. Optimizations Enabled by DBI

We demonstrate the effectiveness of DBI by describing efficient implementations of three cache optimizations: 1) DRAM-aware writeback, 2) cache lookup bypass, and 3) ECC overhead reduction. Note that while prior works have proposed these optimizations [23, 27, 30, 33, 44, 51, 58, 59], DBI has two key advantages over prior proposals. First, implementing these optimizations using DBI is simpler and more efficient than prior works. Second, while combining prior proposals is either impossible or further increases complexity, DBI can simultaneously enable all three optimizations (and many more described in Section 7). Our evaluations (Section 6) show that DBI performs significantly better than any individual optimization, while also reducing the overall area cost. We now describe the three optimizations in detail.

3.1. Efficient DRAM-Aware Aggressive Writeback

The first optimization is referred to as DRAM-Aware Writeback. This optimization is based on two observations. First, each DRAM bank has a structure called the row buffer that caches the last accessed row from that bank [26]. Requests to the row buffer (row buffer hits) are much faster and more efficient than other requests (row buffer misses) [26, 45, 60]. Second, the memory controller buffers writes to DRAM in a write buffer and flushes the writes when the buffer is close to full [27]. Based on these observations, filling the write buffer with blocks from the same DRAM row will lead to a faster and more efficient writeback phase.

Unfortunately, the write sequence to DRAM primarily depends on the order in which dirty blocks are evicted from the LLC. Since blocks of a DRAM row typically map to different cache sets, dirty blocks of the same row are unlikely to be evicted together. As a result, writing back blocks in the order in which they are evicted will cause a majority of writes to result in row misses [27]. To address this problem, a recent work [27] proposed a mechanism to proactively write back all dirty blocks of a DRAM row when any dirty block from that row is evicted from the cache. However, this requires multiple tag store lookups to identify if blocks of the row are dirty. Many of these lookups may be unnecessary as the blocks may actually *not* be dirty or may not even be in the cache.

Virtual Write Queue (VWQ) [51] proposed to address this problem by using a Set State Vector (SSV) to filter some of such unnecessary lookups. The SSV indicates if each cache set has any dirty block in the LRU ways. VWQ looks up a set for dirty blocks only if the SSV indicates that the set has dirty blocks in the LRU ways. Since VWQ only checks the LRU ways to generate proactive writebacks, we find that it is not significantly more efficient compared to DRAM-Aware Writeback [27]. Our evaluations show that, as a result of the additional tag lookups, both DAWB and VWQ perform significantly more tag lookups than the baseline (1.95x and 1.88x respectively—Section 6.1).

In contrast to these mechanisms, implementing such a proactive writeback scheme is straightforward using DBI. When a block is evicted from the cache, the cache first consults the DBI to find out if the block is dirty. If so, the bit vector of the corresponding DBI entry also provides the list of all dirty blocks of the same row. In addition to generating a writeback for the evicted block, the cache also generates writebacks for the other dirty blocks of the row. We call this scheme the *Aggressive Writeback* (AWB) scheme.

Figure 3 pictorially shows the operation of AWB. As shown, AWB looks up the tag store *only for blocks that are actually dirty*, thereby reducing the contention for the cache port. This reduced contention enables AWB to further improve performance compared to prior approaches [27, 51], especially in multi-core systems, where a tag lookup in the shared cache can delay requests from all applications (Section 6.2).[4]

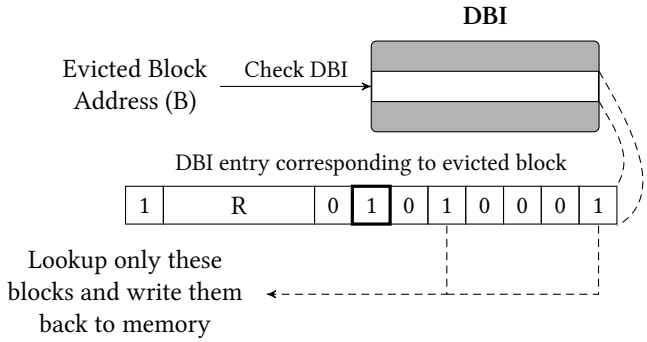


Figure 3: Aggressive writeback using DBI. Block B (the evicted block) is the second block of the DRAM row R and is dirty.

3.2. Efficient Cache Lookup Bypass

The second optimization is based on a simple idea: if an access to the cache is likely to miss, we can potentially bypass the cache lookup, thereby reducing the overall latency and energy consumed by the access [33, 44]. This optimization requires a mechanism to predict whether an access will hit or miss in the cache. The key challenge in realizing this optimization is that when the cache contains dirty blocks, the mechanism cannot bypass the lookup for a block that is dirty in the cache.

Prior works address this problem by either using a write-through cache [44] (no dirty blocks to begin with) or by ensuring that the prediction mechanism has no false positives (no access is falsely predicted to miss in the cache). Both of these approaches have shortcomings: using the write-through policy can significantly increase the memory write bandwidth requirement, and ensuring that there are no false positives in the miss prediction requires complex hardware structures [33].

In contrast to the above approaches, using DBI to maintain the dirty block information enables a simpler approach to bypassing cache lookups that works with *any* prediction

[4] We always prioritize a demand lookup over a lookup for generating aggressive writeback (similar to prior work [27]). However, once started, a lookup cannot be preempted and can delay other queued requests.

mechanism. Our approach is based on the fact that the DBI is much smaller than the main tag store and hence, has lower latency and energy per access. Figure 4 pictorially depicts our proposed mechanism. As shown, our mechanism uses a *miss predictor* to predict if each read access misses in the cache. If an access is predicted to miss in the cache, our mechanism checks the DBI to determine if the block is dirty in the cache. If so, the block is accessed from the cache. Otherwise, the access is forwarded to the next level of the hierarchy. Our mechanism does not modify any of the other cache operations. We refer to this optimization as *Cache Lookup Bypass* (CLB).

CLB can be used with any miss predictor [33, 44, 49, 57]. In our evaluations (Section 6), we employ CLB with the predictor used by Skip Cache [44] as it is simpler to implement and has lower hardware overhead compared to other miss predictors (e.g., [33, 49]). Skip Cache divides execution into epochs and monitors the miss rate of each application/thread in each epoch (using set sampling [41]). If an application's miss rate exceeds a threshold (0.95, in our experiments), all accesses of the application (except those that map to the sampled sets) are predicted to miss in the cache in the next epoch.

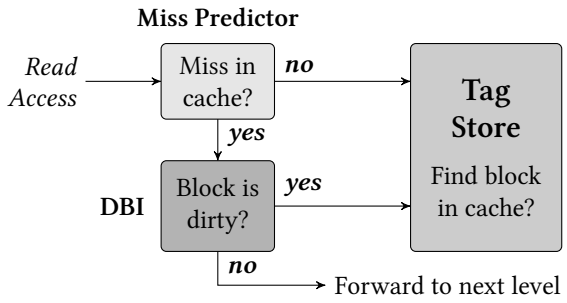


Figure 4: Cache lookup bypass mechanism using DBI

3.3. Reducing ECC Overhead

The third optimization is an approach to reduce the area overhead of Error-Correction Codes (ECC) employed in many processors. With small feature sizes, data reliability is a major concern in modern processors [9]. Therefore, to ensure the integrity of the data in the on-chip caches, processors store ECC (e.g., SECDED – Single Error Correction Double Error Detection code) along with each cache block. However, storing such ECC information comes with an area cost.

This optimization to reduce ECC overhead is based on a simple observation: only dirty blocks require a strong ECC; clean blocks only require error detection. This is because, if an error is detected in a clean block, the block can be retrieved from the next level of the memory hierarchy. On the other hand, if an error is detected in a dirty block, then the cache should also correct the error as it has the *only* copy of the block. To exploit this heterogeneous ECC requirement for clean and dirty blocks, prior works proposed to use a simple Error Detection Code (EDC) for all blocks while storing the ECC only for dirty blocks in a separate structure [30] or in main memory [58, 59]. While these mechanisms are successful in reducing the ECC overhead, they require significant changes to the error correction logic (such as two-tiered error protection [58, 59]), even though errors might be rare to begin with. In contrast to these prior works, in our proposed cache organization, since the DBI is the authoritative source for determining if a block is dirty, it is sufficient to keep ECC information for only the blocks tracked by the DBI.

Figure 5 compares the ECC organization of the baseline cache and a cache augmented with DBI. While the baseline stores ECC for all blocks in the tag store, our mechanism stores *only EDC for all blocks* and stores *additional ECC information for only blocks tracked by the DBI* . As our evaluations show (Section 6), with the two previously discussed optimizations (AWB and CLB), DBI significantly improves performance while tracking far fewer blocks than the main tag store. Therefore, maintaining ECC for only blocks tracked by the DBI significantly reduces the area overhead of ECC without significant additional complexity.

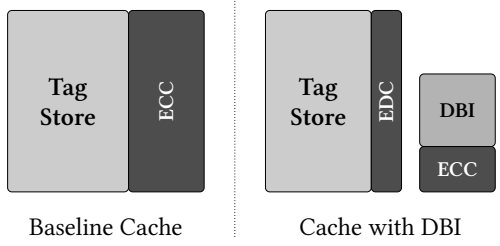


Figure 5: Reducing ECC overhead using DBI. In this example, the cumulative number of blocks tracked by the DBI is $1/4^{\text{th}}$ the number of blocks tracked by the tag store.

4. DBI Design Choices

The DBI design space can be defined using three key parameters: 1) *DBI size*, 2) *DBI granularity* and 3) *DBI replacement policy*.[5] These parameters determine the effectiveness of the three optimizations discussed in the previous section. We now discuss these parameters and their trade-offs in detail.

4.1. DBI Size

The DBI size refers to the cumulative number of blocks tracked by all the entries in the DBI. For ease of analysis across systems with different cache sizes, we represent the DBI size as the ratio of the cumulative number of blocks tracked by the DBI and the number of blocks tracked by the cache tag store. We denote this ratio using α. For example, for a 1MB cache with a 64B block size (16k blocks), a DBI of size $\alpha = 1/2$ enables the DBI to track 8k blocks.

The DBI size presents a trade-off between the size of write working set (set of frequently written blocks) that can be captured by the DBI, and the area, latency, and power cost of the

[5] Similar to the main tag store, DBI is also a set-associative structure and has a fixed associativity. However, we do not discuss the DBI associativity in detail as its trade-offs are similar to any other set-associative structure.

DBI. A large DBI has two benefits: 1) it can track a larger write working set, thereby reducing the writeback bandwidth demand, and 2) it gives more time for a DBI entry to accumulate writebacks to a DRAM row, thereby better exploiting the AWB optimization. However, a large DBI comes at a higher area, latency and power cost. On the other hand, a smaller DBI incurs lower area, latency and power cost. This has two benefits: 1) lower latency in the critical path for the CLB optimization and 2) ECC storage for fewer dirty blocks. However, a small DBI limits the number of dirty blocks in the cache and thus, result in premature DBI evictions, reducing the potential to generate aggressive writebacks. It can also potentially lead to thrashing if the write working set is significantly larger than the number of blocks tracked by the small DBI.

4.2. DBI Granularity

The DBI granularity refers to the number of blocks tracked by a single DBI entry. Although our discussion in Section 2.1 suggests that this is same as the number of blocks in each DRAM row, we can design the DBI to track fewer blocks in each entry. For example, for a system with DRAM row of size 8KB and cache block of size 64B, a natural choice for the DBI granularity is 8KB/64B = 128. Instead, we can design a DBI entry to track only 64 blocks, i.e. one half of a DRAM row.

The DBI granularity presents another trade-off between the amount of locality that can be extracted during the writeback phase (using the AWB optimization) and the size of write working set that can be captured using the DBI. A large granularity leads to better potential for exploiting the AWB optimization. However, if writes have low spatial locality, a large granularity will result in inefficient use of the DBI space, potentially leading to write working set thrashing.

4.3. DBI Replacement Policy

The DBI replacement policy determines which entry is evicted on a DBI eviction, described in Section 2.2.4. A DBI eviction *only writes back* the dirty blocks of the corresponding DRAM row to main memory, and does not evict the blocks themselves from the cache. Therefore, a DBI eviction does not affect the latency of future read requests for the corresponding blocks. However, if the previous cache level generates a writeback request for a block written back due to a DBI eviction, the block will have to be written back to memory again, leading to an additional write to main memory. Therefore, the goal of the DBI replacement policy is to ensure that blocks are not prematurely written back to main memory.

The ideal DBI replacement policy is to evict the DBI entry that has a writeback request farthest into the future. However, similar to Belady's optimal replacement policy for caches [10], this ideal policy is impractical to implement in real systems. We evaluated five practical replacement policies for DBI: 1) Least Recently Written (LRW)—similar to the LRU policy for caches, 2) LRW with Bimodal Insertion Policy [42], 3) Rewrite-interval prediction policy—similar to the RRIP policy for caches [19], 4) Max-Dirty—entry with the maximum number of dirty blocks, 5) Min-Dirty—entry with the minimum number of dirty blocks. We find that the LRW policy works comparably or better than the other policies and use it for all our evaluations in the paper.

5. Evaluation Methodology

System. We use an in-house event-driven x86 multi-core simulator that models out-of-order cores, coupled with a DDR3 [20] DRAM simulator. The simulator faithfully models all processor stalls due to memory accesses. All the simulated systems use a three-level cache hierarchy. The L1 data cache and the L2 cache are private to each core. The L3 cache is shared across all cores. All caches uniformly use a 64B cache block size. We do not enforce inclusion in any level of the cache hierarchy. We implement DBI and prior approaches [27, 44, 51] for the aggressive writeback and cache lookup bypass optimizations at the shared last-level cache. We use CACTI [1] to model the area, latency, and power for the caches and the DBI. Table 1 lists the main configuration parameters in detail.

Benchmarks and Workloads. Our evaluations use benchmarks from the SPEC CPU2006 suite [3] and STREAM [4]. We use Pinpoints [38] to collect instruction traces from the representative portion of these benchmarks. We run each benchmark for 500 million instructions—the first 200 million instructions for warmup and the remaining 300 million instructions to collect the results. For our multi-core evaluations, we classify the benchmarks into nine categories based on read intensity (low, medium, and high) and write intensity (low, medium, and high), and generate multi-programmed workloads with varying levels of read and write intensity. The read intensity of a workload determines how much a workload can get affected by interference due to writes and the write intensity determines how much interference a workload is likely to cause to read accesses. In all, we present results for 102 2-core, 259 4-core, and 120 8-core workloads.

Metrics. We use instruction throughput to evaluate single-core performance, and weighted speedup [50] to evaluate multi-core performance. We also present a detailed analysis of our single-core experiments using other statistics (e.g., MPKI, read/write row hit rates). For multi-core workloads, we present other performance and fairness metrics—instruction throughput, harmonic speedup [32], and maximum slowdown [14, 24]. Weighted and harmonic speedup were shown to correlate with system throughput and response time [15].

6. Results

We evaluate nine different mechanisms, including the baseline, four prior approaches (Dynamic Insertion Policy [18, 42], Skip Cache [44], DRAM-aware Writeback [27], and Virtual Write Queue [51]), and four variants of DBI with different combinations of the two performance optimizations (AWB and CLB). Table 2 lists all the mechanisms, the labels used

Processor	1-8 cores, 2.67 GHz, Single issue, Out-of-order, 128 entry instruction window
L1 Cache	Private, 32KB, 2-way set-associative, tag store latency = 2 cycles, data store latency = 2 cycles, parallel tag and data lookup, LRU replacement policy, number of MSHRs = 32
L2 Cache	Private, 256KB, 8-way set-associative, tag store latency = 12 cycles, data store latency = 14 cycles, parallel tag and data lookup, LRU replacement policy
L3 Cache	Shared, 2MB/core. 1/2/4/8-core, 16/32/32/32-way set-associative, tag store latency = 10/12/13/14 cycles, data store latency = 24/29/31/33 cycles, serial tag and data lookup, LRU replacement policy
DBI	Size (α) = $1/4$, granularity = 64, associativity = 16, latency = 4 cycles, LRW replacement policy (Section 4.3)
DRAM Controller	Open row, row interleaving, FR-FCFS scheduling policy [45, 60], 64-entry write buffer, drain when full policy [27]
DRAM and Bus	DDR3-1066 MHz [20], 1 channel, 1 rank, 8 banks, 8B-wide data bus, burst length = 8, 8KB row buffer

Table 1: Main configuration parameters used for our evaluation

for them in our evaluations, and the values for their key design parameters. All mechanisms except the baseline use TA-DIP [42] to determine the insertion policy for the incoming cache blocks. We do not present detailed results for Skip Cache in our figures as Skip Cache performs comparably to or worse than the baseline/TA-DIP (primarily because it employs the write-through policy). We evaluate the third optimization enabled by DBI—reducing ECC overhead—separately in Section 6.3 as it has no first order effect on performance.

6.1. Single-Core Results

Figure 6 presents the IPC results for our single core experiments. The figure also plots the memory write row hit rate, LLC tag lookups per kilo instructions, memory writes per kilo instructions, and memory read row hit rate for all the benchmarks to illustrate the underlying trends that result in the performance improvement of various mechanisms. For clarity, the figure does not show results for the Baseline LRU policy and for benchmarks with LLC MPKI < 1 or Baseline IPC > 0.9. For these benchmarks, there is no (negative) impact on performance due to any of the mechanisms. We draw several conclusions from our single core results.

First, DBI+AWB significantly outperforms TA-DIP (13% on average), and performs similarly to DAWB and VWQ for almost all benchmarks (Figure 6a). This performance improvement is due to the significant increase in the memory write row hit rate—DAWB, VWQ, and DBI+AWB write back blocks of the same DRAM row together. These mechanisms improve write row hit rate from 35% (TA-DIP) to 88%, 82% and 81%, respectively (Figure 6b).

Second, the key difference between DBI+AWB and the other two mechanisms is brought out by the number of tag lookups they perform. Figure 6c plots the number of tag lookups per kilo instructions for all the mechanisms. DAWB significantly increases the number of tag lookups compared to TA-DIP (by 1.95x on average). This is because, when a dirty block is evicted from the cache, DAWB indiscriminately looks up the tag store for *every* other block of the corresponding DRAM row to check if each block is dirty, leading to a significant number of unnecessary lookups for blocks that are not dirty. As described in Section 3.1, although VWQ aims to reduce the number of unnecessary tag lookups, by checking only the LRU ways for dirty blocks on each dirty eviction, it performs such additional lookups multiple times. Hence, it is not much more effective compared to DAWB (1.85x more tag store lookups compared to TA-DIP). In contrast to both DAWB and VWQ, DBI looks up the tag store for *only* blocks that are actually dirty. As a result, DBI (with AWB) obtains the benefits of aggressive DRAM-aware writeback without increasing tag store contention. Although this reduction in contention does not translate to single-core performance over DAWB and VWQ, DBI+AWB significantly improves multi-core per-

Mechanism	Description
Baseline	Baseline cache using the Least Recently Used (LRU) replacement policy
TA-DIP	Thread-aware dynamic insertion policy [18], 32 dueling sets, 10-bit policy selector, bimodal insertion probability = $\frac{1}{64}$
DAWB	Cache using the DRAM-aware writeback policy [27] and the TA-DIP policy [18] for read accesses
VWQ	Virtual Write Queue [51], cache employs TA-DIP [18]
Skip Cache	Per-application lookup bypass [44], cache employs TA-DIP [18], threshold = 0.95, epoch length = 50 million cycles
DBI	Plain DBI without any optimizations, cache employs TA-DIP [18], DBI parameters listed in Table 1
DBI+AWB	DBI with the aggressive writeback optimization (described in Section 3.1)
DBI+CLB	DBI with the cache lookup bypass optimization (described in Section 3.2), same miss predictor as Skip Cache [44]
DBI+AWB+CLB	DBI with both the aggressive writeback and the cache lookup bypass optimizations

Table 2: List of evaluated mechanisms

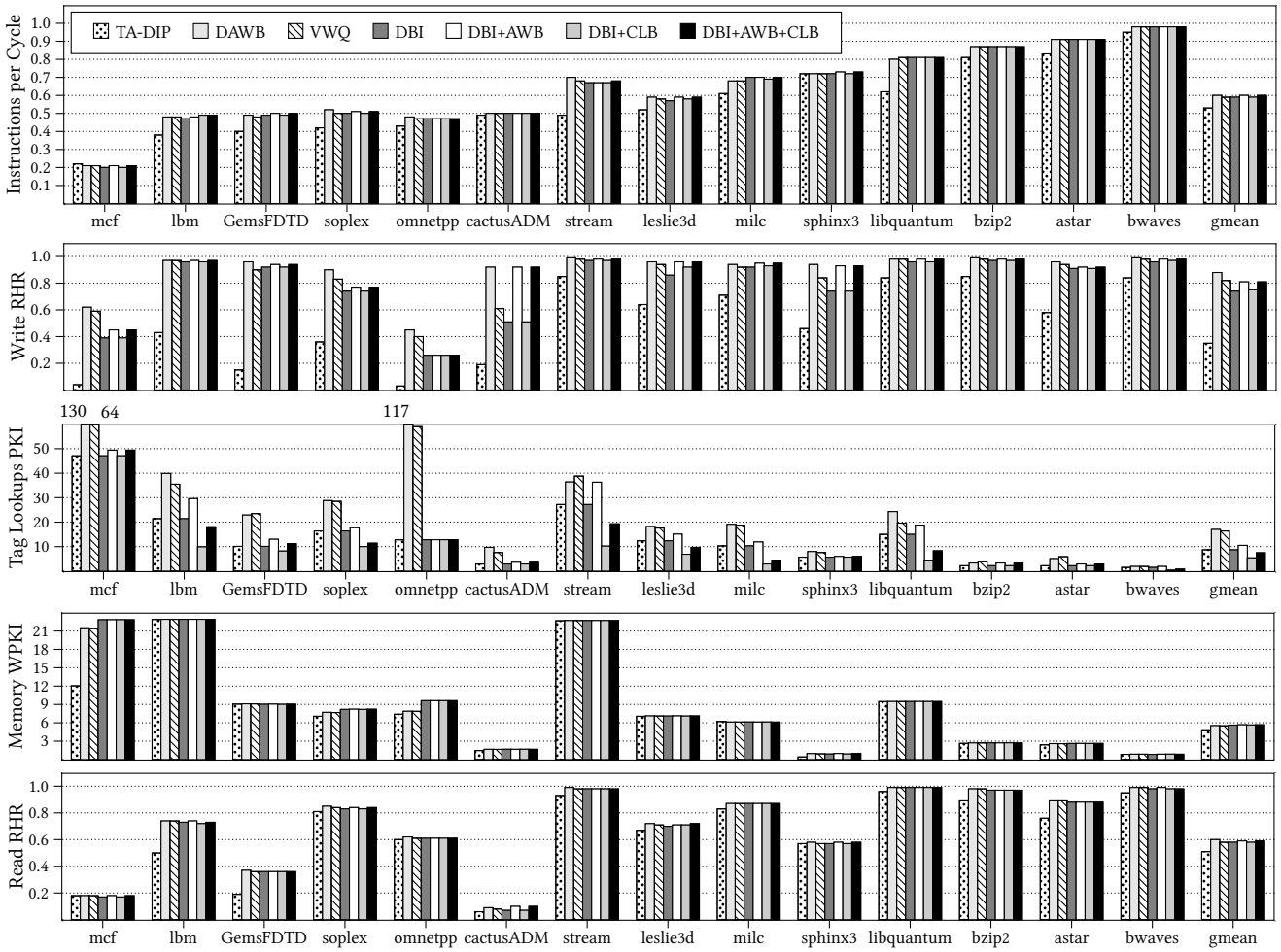


Figure 6: (top to bottom) a) Instructions per cycle (IPC), b) Write Row Hit Rate (Write RHR), c) LLC tag lookups per kilo instructions (Tag lookups PKI), d) Memory Writes per Kilo Instructions (WPKI), e) Read Row Hit Rate (Read RHR). The benchmarks on the x-axis are sorted based on increasing order of baseline IPC.

formance compared to DAWB and VWQ (Section 6.2).

Third, Figure 6c also illustrates the benefits of the cache lookup bypass (CLB) optimization. For applications that do not benefit from the LLC, CLB avoids the tag lookup for accesses that will likely miss in the cache. As a result, the CLB optimization significantly reduces the number of tag lookups for several applications compared to TA-DIP (14% on average). Although this reduction in the number of tag lookups does not improve single core performance, it significantly improves the performance of multi-core systems (Section 6.2).

Fourth, although one may expect DBI (and its variants) to increase the number of writebacks to main memory (due to its proactive writeback mechanism), this is not the case. Even with the aggressive writeback optimization (AWB) enabled, with the exception of *mcf* and *omnetpp*, DBI does not have any visible impact on the number of memory writes per kilo instruction (Figure 6d—Memory WPKI). Although not shown in the figure, DBI and its variants have no impact on the LLC MPKI (read misses per kilo instructions) compared to the TA-DIP policy. This is expected as these mechanisms do not change the cache replacement policy for read requests—they only proactively write back the dirty blocks.

Finally, as a side effect of the increase in write row hit rate, DBI (and its variants) also increase the read row hit rate (Figure 6e). This is because, with a high write row hit rate, very few rows are deactivated during the writeback phase of the memory controller. As a result, many DRAM rows opened by read requests are likely to remain open when the memory controller moves back to the read phase.

In summary, our final mechanism, DBI+AWB+CLB, which combines both the optimizations, provides the best single-core performance compared to all other mechanisms while significantly reducing the number of tag lookups.

6.2. Multi-Core Results

Figure 7 plots the average weighted speedup of different mechanisms for 2-core, 4-core and 8-core systems. We do not show results for VWQ as DAWB performs better than VWQ. The key takeaway from the figure is that DBI with both AWB and CLB optimizations provides the best system throughput

across all mechanisms for all the systems (31% better than the baseline and 6% better than DAWB for 8-core systems).

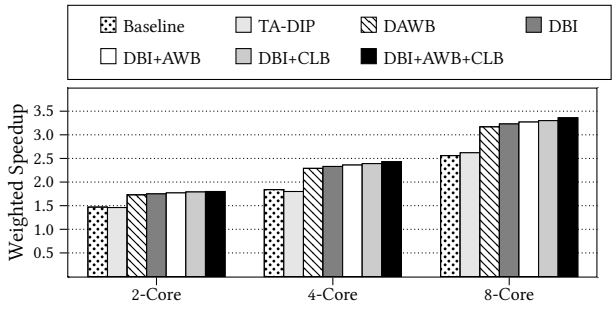


Figure 7: Multi-core system performance

We draw four other conclusions. First, across the many multi-core workloads, TA-DIP performs similarly to the baseline LRU on average. This is because TA-DIP makes suboptimal decisions for some workloads, offsetting the benefit gained in other workloads. Second, DBI by itself performs slightly better than DAWB for all three multi-core systems. This is because, in contrast to DAWB, DBI evictions provide the benefit of DRAM-aware writeback without increasing contention for the tag store. Third, adding the AWB optimization further improves performance by more aggressively exploiting the DRAM-aware writebacks. Across 120 8-core workloads, DBI+AWB improves performance by 3% compared to DAWB. Fourth, the CLB optimization further reduces the contention for tag store lookups. This reduction in contention enables CLB to further improve the performance of DBI (with and without the AWB optimization).

To better understand the benefits of reducing tag store contention, let us consider a case study of the 2-core system running the workload *GemsFDTD* and *libquantum*. For this workload, DAWB performs significant number of unnecessary tag store lookups (2.2x increase in tag store lookups compared to baseline for *GemsFDTD*—Figure 6c). On the other hand, the CLB optimization significantly reduces the number of tag store lookups (3x reduction in tag store lookups for *libquantum* compared to baseline). As a result of reducing contention for tag store, we expect DBI to perform significantly better than DAWB and the CLB optimization to further improve performance. DAWB improves performance compared to baseline by 40%. DBI (even without any optimization) improves performance by 83% compared to baseline (30% compared to DAWB). This is because the DBI evictions obtain the benefit of DRAM-aware writebacks, even without AWB. In fact, enabling the AWB optimization does not buy much performance for this workload (< 1%). Enabling the CLB optimization, as expected, further improves performance (92% compared to baseline and 37% compared to DAWB).

Figure 8 compares the system performance of DAWB with DBI+AWB+CLB for all the 259 4-core workloads. The workloads are sorted based on the weighted speedup improvement of DBI+AWB+CLB. There are two key takeaways from the figure. First, the average performance improvement of DBI over AWB is not due to a small set of workloads. Rather, except for a few workloads, DBI+AWB+CLB consistently outperforms DAWB. Second, DBI slightly degrades performance compared to the baseline only for 7 workloads (mainly due to the additional writebacks generated by DBI).

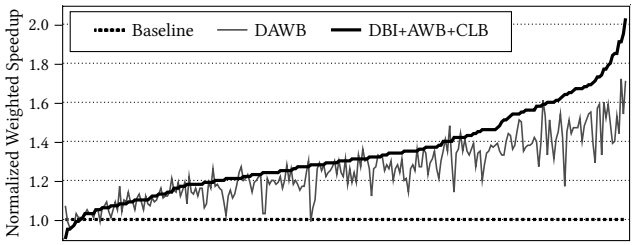


Figure 8: 4-core system performance for all 259 workloads

Table 3 presents three other multi-core performance and fairness metrics: instruction throughput, harmonic speedup [32], and maximum slowdown [14, 24, 25]. As our results indicate, DBI+AWB+CLB improves both system performance and fairness for all multi-core systems.

Number of Cores	2	4	8
Number of workloads	102	259	120
Weighted Speedup [50] Improvement	22%	32%	31%
Instruction Throughput Improvement	23%	32%	30%
Harmonic Speedup [32] Improvement	23%	36%	35%
Maximum Slowdown [14, 24] Reduction	18%	29%	28%

Table 3: Multi-core: Performance and fairness of DBI with both AWB and CLB optimizations compared to baseline

Based on our performance results, we conclude that DBI is a simple and effective mechanism to concurrently enable multiple optimizations that significantly improve system performance. In the next section, we show that DBI achieves this while reducing overall cache area cost.

6.3. Area and Power Analysis (with and without ECC)

A cache augmented with DBI can reduce cache area cost compared to conventional organization due to two reasons. First, DBI tracks far fewer blocks than the main tag store, thereby directly reducing the storage cost for dirty bits. Even when tracking only $1/4^{\text{th}}$ as many blocks as the main tag store, DBI significantly improves performance (as shown in Section 6.2). Second, as we discussed in Section 3.3, DBI can enable a simple mechanism to reduce the ECC overhead, further reducing the cache area cost.

In our evaluations, we assume the ECC organization shown in Figure 5. The baseline cache stores SECDED ECC (12.5% overhead) for all cache blocks, whereas the cache augmented with DBI stores only parity EDC (1.5% overhead) for all cache blocks and SECDED ECC for only blocks tracked by the DBI.

Note that our performance evaluations do not include any overhead of computing EDC or ECC.

Table 4 summarizes the reduction in the tag store and overall cache storage cost due to DBI. Since we scale the DBI size with the cache size, the storage savings of using DBI, in terms of the total number of bits, is roughly independent of the cache size. As highlighted in the table, with $\alpha = 1/4$ and with ECC enabled, DBI reduces the tag store cost (in terms of number of bits) by 44% and the overall cache by 7%.

DBI Size (α)	Without ECC		With ECC	
	Tag Store	Cache	Tag Store	Cache
$1/4$	2%	0.1%	**44%***	**7%**
$1/2$	1%	0.0%	26%*	4%

Table 4: Bit storage cost reduction of cache with DBI compared to conventional cache (with and without ECC). *We assume ECC is stored in the main tag store (or DBI)

We perform a more detailed area and power analysis of a cache with DBI using CACTI [1]. Our results show that the reduction in bit storage cost translates to commensurate reduction in the overall cache area. For a 16MB cache with ECC protection, our mechanism reduces the overall cache area by 8% and 5% for $\alpha = 1/4$ and $\alpha = 1/2$, respectively. We expect the gains to increase with stronger ECC, which will be likely be required in future systems [7, 9].

Table 5 shows the percentage increase in the static and dynamic power consumption of DBI. DBI leads to a marginal increase in both static and dynamic power consumption of the cache. However, our analysis using DDR3 SDRAM System-Power Calculator [2] shows that as a result of increasing the DRAM row hit rate, our mechanism reduces overall memory energy consumption of single-core system by 14% on average compared to the baseline.

Cache size	2 MB	4 MB	8 MB	16MB
Static	0.12%	0.21%	0.21%	0.22%
Dynamic	4%	1%	1%	2%

Table 5: DBI power consumption (fraction of total cache power)

6.4. Sensitivity to DBI Design Parameters

In Section 4, we described three key DBI design parameters that can potentially affect the effectiveness of the different optimizations: 1) DBI granularity, 2) DBI size (α), and 3) DBI replacement policy. As briefly mentioned in that section, the Least Recently Written (LRW) policy performs comparably or better than the other policies. In this section, we evaluate the sensitivity of DBI performance to the DBI granularity and size, which are more unique to our design. We individually analyze the sensitivity of the two performance optimizations, AWB and CLB, to these parameters.

Table 6 shows the sensitivity of the AWB optimization to DBI size and granularity. The table shows the average IPC improvement of DBI+AWB compared to baseline for the single-core system. As expected, the performance of AWB increases with increasing granularity and size. However, a smaller DBI size enables more reduction in the ECC area overhead (as shown in Section 6.3), presenting a trade-off between performance and area.

Granularity		16	32	64	128
Size	$\alpha = 1/4$	10%	12%	12%	13%
	$\alpha = 1/2$	10%	12%	13%	14%

Table 6: Sensitivity of AWB to DBI size and granularity. Values show the average IPC improvement of DBI+AWB compared to baseline for our single-core system.

The effectiveness of the CLB optimization depends on the effectiveness of the miss predictor and the latency of looking up the DBI (Section 3.2). As described in that section, the effectiveness of our miss predictor, Skip Cache, depends on the epoch length and the miss threshold. The access latency of the DBI primarily depends on the size of the DBI. We ran extensive simulations analyzing the sensitivity of the CLB optimization to these parameters. For reasonable values of these parameters (bypass threshold—0.5 to 0.95, epoch length—10 million cycles to 250 million cycles, and DBI size—0.25 to 0.5), we did not find a significant difference in performance improvement of the CLB optimization.

6.5. Sensitivity to Cache Size and Replacement Policy

Table 7 plots the improvement in weighted speedup of DBI (with both AWB and CLB) compared to the Baseline with two different cache sizes for each of the multi-core systems (2MB/Core and 4MB/core). As expected, the performance improvement of DBI decreases with increasing cache size, as memory bandwidth becomes less of an issue with larger cache sizes. However, DBI significantly improves performance compared to the baseline even for large caches (25% for 8-core systems with a 32MB cache).

Cache Size	2-Core	4-Core	8-Core
2MB/Core	22%	32%	31%
4MB/Core	20%	27%	25%

Table 7: Effect of varying cache size. Values indicate performance improvement of DBI+AWB+CLB over baseline.

Since DBI modifies only the writeback sequence of the cache, it does not affect the read hit rate. As a result, we expect the benefits of DBI to complement any benefits from an improved replacement policy. As expected, even when using a better replacement policy, Dynamic Re-reference Interval Prediction policy (DRRIP) [19], DBI significantly improves performance (7% over DAWB for 8-core systems).

7. Other Optimizations Enabled by DBI

Although we have quantitatively evaluated only three optimizations enabled by DBI, there are several other applications for DBI. Our approach can also be employed at other cache levels to organize the dirty bit information to cater to the write access pattern favorable to each cache level—similar to the DRAM row oriented organization in our proposal. In this section, we list a few other potential applications of DBI.

Load Balancing Memory Accesses. A recent prior work [49] proposed a mechanism to load balance memory requests between an on-chip DRAM cache and off-chip memory to improve bandwidth efficiency. For this purpose, they use costly special structures: a counting Bloom filter [16] to keep track of heavily written pages and a small cache to keep track of pages that are likely dirty in the DRAM cache. In our proposed organization, DBI can seamlessly serve both purposes—it can track heavily written pages using its replacement policy and can track pages that are dirty in the cache.

Fast Lookup for Dirty Status. DBI can efficiently answer queries of the nature "Does DRAM row R have any dirty blocks?", "Does DRAM bank/rank X have any dirty blocks?", etc. As a result, DBI can be used more effectively with many opportunistic memory scheduling algorithms that schedule writes based on rank idle time [55], with reads to the same row [21], or other eager writeback mechanisms [28, 35, 48].

Cache Flushing. In many scenarios, large portions of the cache should be flushed—e.g., powering down banks to save power [6, 11], persistent memory updates [13]. In such cases, current systems have to writeback dirty blocks in a brute force manner—by looking up the tag store. In contrast, DBI, with its compact representation of dirty bit information, can improve both the efficiency and latency of such cache flushes.

Direct Memory Access (DMA). DMA operations are often performed in bulk to amortize software overhead. A recent work [47] also proposed a memory-to-memory DMA mechanism to accelerate bulk copy operations. When a device reads data from memory, the memory controller must ensure that the data is not dirty in the cache [5]. DBI can accelerate this coherence operation, especially in case of a bulk DMA where a single DBI query can provide the dirty status of several blocks involved in the DMA.

Metadata about Dirty Blocks. DBI provides a compact, flexible framework that enables the cache to store information about dirty blocks. While we demonstrate the benefit of this framework by reducing the overhead of maintaining ECC, one can use DBI in other similar scenarios which require the cache to store information only about dirty blocks (e.g., compression information in main memory compression [39]).

8. Related Work

To our knowledge, this is the first work that proposes a different way of organizing the dirty bit information of the cache to enable several dirty-block-related optimizations. We have already provided comparisons to prior works [23, 27, 29, 30, 33, 44, 51, 58, 59] that have explored these optimizations. In this section, we discuss other related work.

Loh and Hill [31] proposed a data structure called MissMap with the aim reducing the latency of a cache miss by avoiding the need to look up the tag store (maintained in a DRAM cache). DBI can be favorably combined with MissMap to accelerate costly MissMap entry evictions [31].

Wang et al. [54] propose a mechanism to predict last writes to cache blocks to improve writeback efficiency. This mechanism can be combined with DBI to eliminate premature aggressive writebacks.

Khan et al. propose a mixed-cell architecture [22]—most of the cache built with small cells and a small portion built with larger, more reliable cells. By storing dirty blocks in the more reliable portion of the cache, the amount and strength of ECC required for the cache can be reduced. This approach, although effective, limits the number of dirty blocks in each set, thereby increasing the number of writebacks. DBI, in contrast, reduces ECC overhead without requiring *any* changes to the cache data store design. Having said that, DBI can be combined with such mixed-cell designs to enable all the other optimizations discussed in Sections 3 and 7.

Several prior works (e.g., [12, 17, 19, 40, 43, 46, 53, 56]) have proposed efficient cache management strategies to improve overall system performance. Our proposed optimizations can be used to both improve the writeback efficiency (AWB) and bypass the cache altogether for read accesses of applications (CLB) for which these cache management strategies are not effective. Similarly, AWB can also be combined with different memory read scheduling algorithms (e.g., [8, 24, 25, 34, 36]).

9. Conclusion

We presented the Dirty-Block Index (DBI), a new and simple structure that decouples the dirty bit information from the main cache tag store. This decoupling allows the DBI to organize the dirty bit information independent of the tag store. More specifically, DBI 1) groups the dirty bit information of blocks in the same DRAM row in the same DBI entry, and 2) tracks far fewer blocks than the main tag store. We presented simple and efficient implementations of three optimizations related to dirty blocks using DBI. DBI, with all three optimizations, performs significantly better than individually employing any single optimization (6% better than best previous mechanism for 8-core systems across 120 workloads), while also reducing overall cache area by 8%.

Although we have demonstrated the benefits of DBI using these three optimizations applied to the LLC, this approach is an effective way of enabling several other optimizations at different levels of caches by organizing the DBI to cater to the write patterns of each cache level. We believe this approach can be extended to more efficiently organize other metadata in caches (e.g., cache coherence states), enabling more optimizations to improve performance and power-efficiency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. We thank the members of the SAFARI and LBA research groups for their feedback and the stimulating research environment they provide. We acknowledge the generous support from our industry partners: IBM, Intel, Qualcomm, and Samsung. This research was partially funded by NSF grants (CAREER Award CCF 0953246, CCF 1212962, and CNS 1065112), Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

[1] CACTI 6.0. http://www.cs.utah.edu/~rajeev/cacti6/.

[2] Micron Technologies, Inc., DDR3 SDRAM system-power calculator. http://www.micron.com/products/support/power-calc/.

[3] SPEC CPU2006 Benchmark Suite. www.spec.org/cpu2006.

[4] STREAM Benchmark. http://www.streambench.org/.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, chapter 11, page 12. April 2012.

[6] Software techniques for shared-cache multi-core systems. http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems, 2012.

[7] A. R. Alameldeen et al. Energy-efficient cache design using variable-strength error-correcting codes. In *ISCA*, 2011.

[8] R. Ausavarungnirun et al. Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012.

[9] R. Baumann. Soft errors in advanced computer systems. *IEEE Design Test of Computers*, 22(3):258–266, 2005.

[10] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78 –101, 1966.

[11] K. Chang et al. Enabling efficient dynamic resizing of large DRAM caches via a hardware consistent hashing mechanism. Technical Report Safari TR 2013-001, Carnegie Mellon University, 2013.

[12] J. D. Collins and D. M. Tullsen. Hardware identification of cache conflict misses. In *MICRO*, 1999.

[13] J. Condit et al. Better I/O through byte-addressable, persistent Memory. In *SOSP*, 2009.

[14] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[15] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.

[16] L. Fan et al. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM ToN*, 8(3):281–293, June 2000.

[17] E. G. Hallnor and S. K. Reinhardt. A fully associative software managed cache design. In *ISCA*, 2000.

[18] A. Jaleel et al. Adaptive insertion policies for managing shared caches. In *PACT*, 2008.

[19] A. Jaleel et al. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*, 2010.

[20] JEDEC. DDR3 SDRAM, JESD79-3F, 2012.

[21] M. Jeon. Reducing DRAM row activations with eager writeback. Master's thesis, Rice University, 2012.

[22] S. M. Khan et al. Improving multi-core performance using mixed-cell cache architecture. In *HPCA*, 2013.

[23] S. Kim. Area-efficient error protection for caches. In *DATE*, 2006.

[24] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[25] Y. Kim et al. Thread Cluster Memory Scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[26] Y. Kim et al. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *ISCA*, 2012.

[27] C. J. Lee et al. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-2, University of Texas at Austin, 2010.

[28] H. S. Lee et al. Eager writeback - a technique for improving bandwidth utilization. In *MICRO*, 2000.

[29] L. Li et al. Soft Error and Energy Consumption Interactions : A Data Cache Perspective. In *ISLPED*, 2004.

[30] G. Liu. ECC-Cache: A novel low power scheme to protect large-capacity L2 caches from transient faults. IAS, 2009.

[31] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *MICRO*, 2011.

[32] Kun Luo et al. Balancing thoughput and fairness in SMT processors. In *ISPASS*, 2001.

[33] G. Memik et al. Just Say No: Benefits of early cache miss determination. In *HPCA*, 2003.

[34] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.

[35] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.

[36] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO*, 2006.

[37] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*, 1984.

[38] H. Patil et al. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *MICRO*, 2004.

[39] G. Pekhimenko et al. Linearly Compressed Pages: A low-complexity, low-latency main memory compression framework. In *MICRO*, 2013.

[40] M. K. Qureshi et al. The V-way cache: Demand based associativity via global replacement. In *ISCA*, 2005.

[41] M. K. Qureshi et al. A case for MLP-aware cache replacement. In *ISCA*, 2006.

[42] M. K. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[43] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[44] K. Raghavendra et al. SkipCache: Miss-rate aware cache management. In *PACT*, 2012.

[45] S. Rixner et al. Memory access scheduling. In *ISCA*, 2000.

[46] V. Seshadri et al. The Evicted-Address Filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

[47] V. Seshadri et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *MICRO*, 2013.

[48] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA*, 2007.

[49] J. Sim et al. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *MICRO*, 2012.

[50] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[51] J. Stuecheli et al. The Virtual Write Queue: Coordinating DRAM and last-level cache policies. In *ISCA*, 2010.

[52] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *ISCA*, 1986.

[53] G. Tyson et al. A modified approach to data cache management. In *MICRO*, 1995.

[54] Z. Wang et al. Improving writeback efficiency with decoupled last-write prediction. In *ISCA*, 2012.

[55] Z. Wang and D. A. Jiménez. Exploiting rank idle time for scheduling last-level cache writeback. In *PACT*, 2011.

[56] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.

[57] A. Yoaz et al. Speculation techniques for improving load related instruction scheduling. In *ISCA*, 1999.

[58] D. H. Yoon and M. Erez. Flexible cache error protection using an ECC FIFO. In *SC*, 2009.

[59] D. H. Yoon and M. Erez. Memory mapped ECC: Low-cost error protection for last level caches. In *ISCA*, 2009.

[60] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. US Patent 5630096, 1997.